

AD-A239 812



ACTION PAGE

Form Approved
OPM No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

| | | | | | |
|--|--|--|----------------------------|---|--|
| 1. AGENCY USE ONLY (Leave Blank) | | 2. REPORT DATE | | 3. REPORT TYPE AND DATES COVERED Final: 15 Apr 1991 to 01 Jun 1993 | |
| 4. TITLE AND SUBTITLE Alsys, AlsyCOMP_049, Version 1.83, VAX 8530 (Host) to IDT7R301 System (R3000/R3010 bare)(Target), 91040711.11144 | | | | 5. FUNDING NUMBERS | |
| 6. AUTHOR(S) IABG-AVF Ottobrunn, Federal Republic of Germany | | | | | |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) IABG-AVF, Industrianlagen-Betriebsgesellschaft Dept. SZT/ Einsteinstrasse 20 D-8012 Ottobrunn FEDERAL REPUBLIC OF GERMANY | | | | 8. PERFORMING ORGANIZATION REPORT NUMBER IABG-VSR 091 | |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Ada Joint Program Office United States Department of Defense Pentagon, Rm 3E114 Washington, D.C. 20301-3081 | | | | 10. SPONSORING/MONITORING AGENCY REPORT NUMBER | |
| 11. SUPPLEMENTARY NOTES | | | | | |
| 12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited. | | | | 12b. DISTRIBUTION CODE | |
| 13. ABSTRACT (Maximum 200 words) Alsys, AlsyCoMP_049, Version 1.83, Ottobrunn, Germany, VAX 8530 under VMS, Version 5.3-1 (Host) to Integrated Device Technology IDT7RS301 system (R3000/R3010 bare machine)(Target), ACVC 1.11. | | | | | |
| <div style="display: flex; justify-content: space-between; align-items: center;"> <div style="text-align: center;"> <p>DTIC ELECTE S B D AUG 27 1991</p> </div> <div style="text-align: center;"> <p>81</p> <p>91-08764</p> </div> </div> | | | | | |
| 14. SUBJECT TERMS Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO. | | | | 15. NUMBER OF PAGES | |
| | | | | 16. PRICE CODE | |
| 17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED | 18. SECURITY CLASSIFICATION UNC ASSIFED | 19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED | 20. LIMITATION OF ABSTRACT | | |

91 0 22 081

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 7 April, 1991.

Compiler Name and Version: **AlsysCOMP_049, Version 1.83**

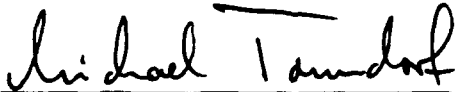
Host Computer System: **VAX 8530 under VMS, Version 5.3-1**

Target Computer System: **Integrated Device Technology IDT7RS301 system. (R3000/R3010 bare machine)**

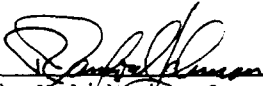
See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 910407I1.11144 is awarded to Alsys. This certificate expires on 1 March, 1993.

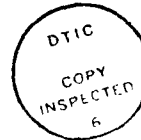
This report has been reviewed and is approved.




IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany



607 Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311





Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

| | |
|--------------------|-------------------------------------|
| Accession For | |
| NTIS GRA&I | <input checked="" type="checkbox"/> |
| DTIC TAB | <input type="checkbox"/> |
| Unannounced | <input type="checkbox"/> |
| Justification | |
| By | |
| Distribution/ | |
| Availability Codes | |
| Dist | Avail and/or Special |
| A-1 | |

AVF Control Number: IABG-VSR 091
15 April 1991

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #910407I1.11144
Alsys
AlsyCOMP_049, Version 1.83
VAX 8530 => IDT7RS301 System
(R3000/R3010 bare)

-- based on TEMPLATE Version 91-01-10 --

Prepared By:
IABG mbH, Abt. ITE
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 7 April, 1991.

Compiler Name and Version: AlsysCOMP_049, Version 1.83


Host Computer System: VAX 8530 under VMS, Version 5.3-1

Target Computer System: Integrated Device Technology IDT7RS301 system. (R3000/R3010 bare machine)

See Section 3.1 for any additional information about the testing environment.


As a result of this validation effort, Validation Certificate 910407I1.11144 is awarded to Alsys. This certificate expires on 1 March, 1993.

This report has been reviewed and is approved.



IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany



for  Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Declaration of Conformance

Customer: Alsys

Ada Validation Facility: IABG mbH Abt. ITE

ACVC Version: 1.11

Ada Implementation:

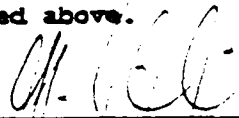
Ada Compiler Name and Version: AlsyCOMP_049, Version 1.83

Host Computer System: VAX 8530 under VMS, Version 5.3-1

Target Computer System: Integrated Device Technology IDT7RS301 System
(R3000/R3010) (bare machine)

Declaration:

[I/we] the undersigned, declare that [I/we] have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.



Customer Signature
Dr. Georg Winterstein

09.04.91

Date

TABLE OF CONTENTS

| | | |
|------------|---|-----|
| CHAPTER 1 | INTRODUCTION | |
| 1.1 | USE OF THIS VALIDATION SUMMARY REPORT | 1-1 |
| 1.2 | REFERENCES | 1-2 |
| 1.3 | ACVC TEST CLASSES | 1-2 |
| 1.4 | DEFINITION OF TERMS | 1-3 |
| CHAPTER 2 | IMPLEMENTATION DEPENDENCIES | |
| 2.1 | WITHDRAWN TESTS | 2-1 |
| 2.2 | INAPPLICABLE TESTS | 2-1 |
| 2.3 | TEST MODIFICATIONS | 2-4 |
| CHAPTER 3 | PROCESSING INFORMATION | |
| 3.1 | TESTING ENVIRONMENT | 3-1 |
| 3.2 | SUMMARY OF TEST RESULTS | 3-1 |
| 3.3 | TEST EXECUTION | 3-2 |
| APPENDIX A | MACRO PARAMETERS | |
| APPENDIX B | COMPILATION SYSTEM OPTIONS | |
| APPENDIX C | APPENDIX F OF THE Ada STANDARD | |

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
- [Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 90.
- [UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

| | |
|---|---|
| Ada Compiler | The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof. |
| Ada Compiler Validation Capability (ACVC) | The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report. |
| Ada Implementation | An Ada compiler with its host computer system and its target computer system. |
| Ada Validation Facility (AVF) | The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation. |
| Ada Validation Organization (AVO) | The part of the certification body that provides technical guidance for operations of the Ada certification system. |
| Compliance of an Ada Implementation | The ability of the implementation to pass an ACVC version. |
| Computer System | A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units. |
| Conformity | Fulfillment by a product, process or service of all requirements specified. |

INTRODUCTION

| | |
|------------------------------|---|
| Customer | An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed. |
| Declaration of Conformance | A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized. |
| Host Computer System | A computer system where Ada source programs are transformed into executable form. |
| Inapplicable test | A test that contains one or more test objectives found to be irrelevant for the given Ada implementation. |
| ISO | International Organization for Standardization. |
| Operating System | Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible. |
| Target Computer System | A computer system where the executable form of Ada programs are executed. |
| Validated Ada Compiler | The compiler of a validated Ada implementation. |
| Validated Ada Implementation | An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90]. |
| Validation | The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation. |
| Withdrawn test | A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language. |

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is March 14, 1991.

| | | | | | |
|---------|---------|---------|---------|---------|---------|
| E28005C | B28006C | C34006D | C35508I | C35508J | C35508M |
| C35508N | C35702A | C35702B | B41308B | C43004A | C45114A |
| C45346A | C45612A | C45612B | C45612C | C45651A | C46022A |
| B49008A | A74006A | C74308A | B83022B | B83022H | B83025B |
| B83025D | B83026B | C83026A | C83041A | B85001L | C86001F |
| C94021A | C97116A | C98003B | BA2011A | CB7001A | CB7001B |
| CB7004A | CC1223A | BC1226A | CC1226B | BC3009B | AD1B08A |
| BD1B02B | BD1B06A | BD2A02A | CD2A21E | CD2A23E | CD2A32A |
| CD2A41A | CD2A41E | CD2A87A | CD2B15C | BD3006A | BD4008A |
| CD4022A | CD4022D | CD4024B | CD4024C | CD4024D | CD4031A |
| CD4051D | CD5111A | CD7004C | ED7005D | CD7005E | AD7006A |
| CD7006E | AD7201A | AD7201E | CD7204B | AD7206A | BD8002A |
| BD8004C | CD9005A | CD9005B | CDA201E | CE2107I | CE2117A |
| CE2117B | CE2119B | CE2205B | CE2405A | CE3111C | CE3116A |
| CE3118A | CE3411B | CE3412B | CE3607B | CE3607C | CE3607D |
| CE3812A | CE3814A | CE3902B | | | |

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Approved Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Approved Ada Commentaries are included as appropriate.

IMPLEMENTATION DEPENDENCIES

The following 201 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

| | |
|---------------------------|-----------------------|
| C24113L..Y (14 tests) (*) | C35705L..Y (14 tests) |
| C35706L..Y (14 tests) | C35707L..Y (14 tests) |
| C35708L..Y (14 tests) | C35802L..Z (15 tests) |
| C45241L..Y (14 tests) | C45321L..Y (14 tests) |
| C45421L..Y (14 tests) | C45521L..Z (15 tests) |
| C45524L..Z (15 tests) | C45621L..Z (15 tests) |
| C45641L..Y (14 tests) | C46012L..Z (15 tests) |

(*) C24113W..Y (3 tests) contain lines of length greater than 255 characters which are not supported by this implementation.

The following 21 tests check for the predefined type `LONG_INTEGER`:

| | | | | |
|---------|---------|---------|---------|---------|
| C35404C | C45231C | C45304C | C45411C | C45412C |
| C45502C | C45503C | C45504C | C45504F | C45611C |
| C45612C | C45613C | C45614C | C45631C | C45632C |
| B52004D | C55B07A | B55B09C | B86001W | C86006C |
| CD7101F | | | | |

C35713B, C45423B, B86001T, and C86006H check for the predefined type `SHORT_FLOAT`.

C35713D and B86001Z check for a predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`.

C41401A checks that `CONSTRAINT_ERROR` is raised upon the evaluation of various attribute prefixes; this implementation derives the attribute values from the subtype of the prefix at compilation time, and thus does not evaluate the prefix or raise the exception. (See Section 2.3.)

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a `SYSTEM.MAX_MANTISSA` of 47 or greater; for this implementation, there is no such type.

C45624A checks that the proper exception is raised if `MACHINE_OVERFLOW` is `FALSE` for floating point types with digits 5. For this implementation, `MACHINE_OVERFLOW` is `TRUE`.

C45624B checks that the proper exception is raised if `MACHINE_OVERFLOW` is `FALSE` for floating point types with digits 6. For this implementation, `MACHINE_OVERFLOW` is `TRUE`.

B86001Y checks for a predefined fixed-point type other than `DURATION`.

C96005B checks for values of type `DURATION'BASE` that are outside the range of `DURATION`. There are no such values for this implementation.

IMPLEMENTATION DEPENDENCIES

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than was specified by the length clause, as allowed by AI-00558.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

The following 264 tests check for sequential, text, and direct access files:

| | | | |
|----------------|----------------|----------------|-----------------|
| CE2102A..C (3) | CE2102G..H (2) | CE2102K | CE2102N..Y (12) |
| CE2103C..D (2) | CE2104A..D (4) | CE2105A..B (2) | CE2106A..B (2) |
| CE2107A..H (8) | CE2107L | CE2108A..H (8) | CE2109A..C (3) |
| CE2110A..D (4) | CE2111A..I (9) | CE2115A..B (2) | CE2120A..B (2) |
| CE2201A..C (3) | EE2201D..E (2) | CE2201F..N (9) | CE2203A |
| CE2204A..D (4) | CE2205A | CE2206A | CE2208B |
| CE2401A..C (3) | EE2401D | CE2401E..F (2) | EE2401G |
| CE2401H..L (5) | CE2403A | CE2404A..B (2) | CE2405B |
| CE2406A | CE2407A..B (2) | CE2408A..B (2) | CE2409A..B (2) |
| CE2410A..B (2) | CE2411A | CE3102A..C (3) | CE3102F..H (3) |
| CE3102J..K (2) | CE3103A | CE3104A..C (3) | CE3106A..B (2) |
| CE3107B | CE3108A..B (2) | CE3109A | CE3110A |
| CE3111A..B (2) | CE3111D..E (2) | CE3112A..D (4) | CE3114A..B (2) |
| CE3115A | CE3119A | EE3203A | EE3204A |
| CE3207A | CE3208A | CE3301A | EE3301B |
| CE3302A | CE3304A | CE3305A | CE3401A |
| CE3402A | EE3402B | CE3402C..D (2) | CE3403A..C (3) |
| CE3403E..F (2) | CE3404B..D (3) | CE3405A | EE3405B |
| CE3405C..D (2) | CE3406A..D (4) | CE3407A..C (3) | CE3408A..C (3) |
| CE3409A | CE3409C..E (3) | EE3409F | CE3410A |

CE3202A assumes that the NAME operation is supported for STANDARD_INPUT and STANDARD_OUTPUT. For this implementation the underlying operating system does not support the NAME operation for STANDARD_INPUT and STANDARD_OUTPUT. Thus the calls of the NAME operation for the standard files in this test raise USE_ERROR.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 18 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

| | | | | | |
|---------|---------|---------|---------|---------|---------|
| B22003A | B24009A | B29001A | B38003A | B38009A | B38009B |
| B91001H | BC2001D | BC2001E | BC3204B | BC3205B | BC3205D |

C34007P and C34007S were graded passed by Evaluation Modification as directed by the AVO. These tests include a check that the evaluation of the selector "all" raises `CONSTRAINT_ERROR` when the value of the object is null. This implementation determines the result of the equality tests at lines 207 and 223, respectively, based on the subtype of the object; thus, the selector is not evaluated and no exception is raised, as allowed by LRM 11.6(7). The tests were graded passed given that their only output from Report.Failed was the message "NO EXCEPTION FOR NULL.ALL - 2".

C41401A was graded inapplicable by Evaluation Modification as directed by the AVO. This test checks that the evaluation of attribute prefixes that denote variables of an access type raises `CONSTRAINT_ERROR` when the value of the variable is null and the attribute is appropriate for an array or task type. This implementation derives the array attribute values from the subtype; thus, the prefix is not evaluated and no exception is raised, as allowed by LRM 11.6(7), for the checks at lines 77, 87, 97, 108, 121, 131, 141, 152, 165, & 175.

BC3204C..D and BC3205C..D (4 tests) were graded passed by Evaluation Modification as directed by the AVO. These tests are expected to produce compilation errors, but this implementation compiles the units without error; all errors are detected at link time. This behavior is allowed by AI-00256, as the units are illegal only with respect to units that they do not depend on.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact in Germany for technical and sales information about this Ada implementation system, see:

Alsys Gmbh & Co. KG
Am Rüppurrer Schloß 7
W-7500 Karlsruhe 51
Germany
Tel. +49 721 883025

For a point of contact outside Germany for technical and sales information about this Ada implementation system, see:

Alsys Inc.
67 South Bedford Str.
Burlington MA
01803-5152
USA
Tel. +617 270 0030

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

| | | |
|--|------|---------|
| a) Total Number of Applicable Tests | 3652 | |
| b) Total Number of Withdrawn Tests | 93 | |
| c) Processed Inapplicable Tests | 53 | |
| d) Non-Processed I/O Tests | 264 | |
| e) Non-Processed Floating-Point Precision Tests | 201 | |
| f) Total Number of Inapplicable Tests | 518 | (c+d+e) |
| g) Total Number of Tests for ACVC 1.11 | 4170 | (a+b+f) |

The above number of I/O tests were not processed because this implementation does not support a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 518 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation and 264 executable tests that use file operations not supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A Magnetic Tape Reel containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system via a V24 connection and run. The results were captured on the host computer system.

PROCESSING INFORMATION

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

Compiler options :

/LOG tells the Compiler to write additional messages onto the specified file.

Linker options :

/BASE together with a parameter specifies the file containing the result of a previous incremental link, on which the final link is to be performed.

/IMAGE together with a parameter specifies the name of the file which will contain the result of the final link.

/LOG tells the implicitly invoked Completer to write additional messages onto the specified file.

/DIRECTIVE together with a parameter specifies the file which contains the linker directives.

Test output, compiler and linker listings, and job logs were captured on a Magnetic Tape Reel and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented below.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The following macro parameters are defined in term of the value V of `$MAX_IN_LEN` which is the maximum input line length permitted for the tested implementation. For these parameters, Ada string expressions are given rather than the macro values themselves.

| Macro Parameter | Macro Value |
|---|---|
| <code>\$BIG_ID1</code> | $(1..V-1 \Rightarrow 'A', V \Rightarrow '1')$ |
| <code>\$BIG_ID2</code> | $(1..V-1 \Rightarrow 'A', V \Rightarrow '2')$ |
| <code>\$BIG_ID3</code> | $(1..V/2 \Rightarrow 'A') \ \& \ '3' \ \& \ (1..V-1-V/2 \Rightarrow 'A')$ |
| <code>\$BIG_ID4</code> | $(1..V/2 \Rightarrow 'A') \ \& \ '4' \ \& \ (1..V-1-V/2 \Rightarrow 'A')$ |
| <code>\$BIG_INT_LIT</code> | $(1..V-3 \Rightarrow '0') \ \& \ "298"$ |
| <code>\$BIG_REAL_LIT</code> | $(1..V-5 \Rightarrow '0') \ \& \ "690.0"$ |
| <code>\$BIG_STRING1</code> | $"' \ \& \ (1..V/2 \Rightarrow 'A') \ \& \ '"$ |
| <code>\$BIG_STRING2</code> | $"' \ \& \ (1..V-1-V/2 \Rightarrow 'A') \ \& \ '1' \ \& \ '"$ |
| <code>\$BLANKS</code> | $(1..V-20 \Rightarrow '')$ |
| <code>\$MAX_LEN_INT_BASED_LITERAL</code> | $"2:" \ \& \ (1..V-5 \Rightarrow '0') \ \& \ "11:"$ |
| <code>\$MAX_LEN_REAL_BASED_LITERAL</code> | $"16:" \ \& \ (1..V-7 \Rightarrow '0') \ \& \ "F.E:"$ |
| <code>\$MAX_STRING_LITERAL</code> | $"' \ \& \ (1..V-2 \Rightarrow 'A') \ \& \ '"$ |

The following table contains the values for the remaining macro parameters.

| Macro Parameter | Macro Value |
|-------------------------|---------------------------------|
| \$MAX_IN_LEN | 255 |
| \$ACC_SIZE | 32 |
| \$ALIGNMENT | 4 |
| \$COUNT_LAST | 2147483647 |
| \$DEFAULT_MEM_SIZE | 2147483648 |
| \$DEFAULT_STOR_UNIT | 8 |
| \$DEFAULT_SYS_NAME | MIPS_BARE |
| \$DELTA_DOC | 2#1.0#E-31 |
| \$ENTRY_ADDRESS | SYSTEM.INTERRUPT_VECTOR(1) |
| \$ENTRY_ADDRESS1 | SYSTEM.INTERRUPT_VECTOR(2) |
| \$ENTRY_ADDRESS2 | SYSTEM.INTERRUPT_VECTOR(3) |
| \$FIELD_LAST | 512 |
| \$FILE_TERMINATOR | ' ' |
| \$FIXED_NAME | NO_SUCH_FIXED_TYPE |
| \$FLOAT_NAME | NO_SUCH_FLOAT_TYPE |
| \$FORM_STRING | "" |
| \$FORM_STRING2 | "CANNOT RESTRICT FILE CAPACITY" |
| \$GREATER_THAN_DURATION | 0.0 |

| | |
|---------------------------------------|---------------------------------|
| \$GREATER_THAN_DURATION_BASE_LAST | 200_000.0 |
| \$GREATER_THAN_FLOAT_BASE_LAST | 16#1.0#E+32 |
| \$GREATER_THAN_FLOAT_SAFE_LARGE | 16#0.8#E+32 |
| \$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE | 0.0 |
| \$HIGH_PRIORITY | 15 |
| \$ILLEGAL_EXTERNAL_FILE_NAME1 | FILE1 |
| \$ILLEGAL_EXTERNAL_FILE_NAME2 | FILE2 |
| \$INAPPROPRIATE_LINE_LENGTH | -1 |
| \$INAPPROPRIATE_PAGE_LENGTH | -1 |
| \$INCLUDE_PRAGMA1 | PRAGMA INCLUDE ("A28006D1.TST") |
| \$INCLUDE_PRAGMA2 | PRAGMA INCLUDE ("B28006F1.TST") |
| \$INTEGER_FIRST | -2147483648 |
| \$INTEGER_LAST | 2147483647 |
| \$INTEGER_LAST_PLUS_1 | 2147483648 |
| \$INTERFACE_LANGUAGE | ASSEMBLER |
| \$LESS_THAN_DURATION | -0.0 |
| \$LESS_THAN_DURATION_BASE_FIRST | -200_000.0 |
| \$LINE_TERMINATOR | ASCII.LF |
| \$LOW_PRIORITY | 0 |

| | |
|--------------------------|---------------------------|
| \$MACHINE_CODE_STATEMENT | NULL; |
| \$MACHINE_CODE_TYPE | NO_SUCH_TYPE |
| \$MANTISSA_DOC | 31 |
| \$MAX_DIGITS | 15 |
| \$MAX_INT | 2147483647 |
| \$MAX_INT_PLUS_1 | 2147483648 |
| \$MIN_INT | -2147483648 |
| \$NAME | SHORT_SHORT_INTEGER |
| \$NAME_LIST | MIPS_BARE |
| \$NAME_SPECIFICATION1 | X2120A |
| \$NAME_SPECIFICATION2 | X2120B |
| \$NAME_SPECIFICATION3 | X3119A |
| \$NEG_BASED_INT | 16#FFFFFFFE# |
| \$NEW_MEM_SIZE | 2147483648 |
| \$NEW_STOR_UNIT | 8 |
| \$NEW_SYS_NAME | MIPS_BARE |
| \$PAGE_TERMINATOR | '' |
| \$RECORD_DEFINITION | NEW INTEGER |
| \$RECORD_NAME | NO_SUCH_MACHINE_CODE_TYPE |

| | |
|----------------------------|------------------------------|
| \$TASK_SIZE | 32 |
| \$TASK_STORAGE_SIZE | 10240 |
| \$TICK | 2.0 ** (-14) |
| \$VARIABLE_ADDRESS | GET_VARIABLE_ADDRESS |
| \$VARIABLE_ADDRESS1 | GET_VARIABLE_ADDRESS1 |
| \$VARIABLE_ADDRESS2 | GET_VARIABLE_ADDRESS2 |
| \$YOUR_PRAGMA | RESIDENT |

APPENDIX B

COMPILATION AND LINKER SYSTEM OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

4 Compiling

After a program library has been created, one or more compilation units can be compiled in the context of this library. The compilation units can be placed on different source files or they can all be on the same file. One unit, a parameterless procedure, acts as the main program. If all units needed by the main program and the main program itself have been compiled successfully, they can be linked. The resulting code can then be executed by the Alsys Ada System commands ADA LOAD and ADA START.

§4.1 and Chapter 5 describe in detail how to call the Compiler and the Linker. In §4.2 the Completer, which is called to generate code for instances of generic units, is described. §4.5 describes the assembly of external units.

Chapter 6 explains the information which is given if the execution of a program is abandoned due to an unhandled exception.

The information the Compiler produces and outputs in the Compiler listing is explained in §4.4.

Finally, the log of a sample session is given in Chapter 7.

4.1 Compiling Ada Units

To start the Alsys Ada Compiler, use the ADA COMPILE command.

| ADA COMPILE | Command Description |
|--------------------------------|-----------------------|
| Format | |
| \$ ADA COMPILE file-spec[,...] | |
| Command Qualifiers | Defaults |
| /[NO]ANALYZE_DEPENDENCY | /NOANALYZE_DEPENDENCY |
| /LIBRARY=directory-spec | /LIBRARY=[.ADALIB] |
| /[NO]LOG[=file-spec] | /NOLOG |
| /[NO]RECOMPILE | /NORECOMPILE |
| Positional Qualifiers | Defaults |
| /[NO]AUTO_INLINE | /AUTO_INLINE |
| /[NO]CHECK | /CHECK |
| /[NO]COPY_SOURCE | /COPY_SOURCE |
| /[NO]INLINE | /INLINE |
| /[NO]LIST[=file-spec] | /NOLIST |
| /[NO]MACHINE_CODE | /NOMACHINE_CODE |
| /[NO]OPTIMIZE=(FULL PARTIAL) | /OPTIMIZE=FULL |

Command Parameters

`file-spec`

Specifies the file(s) to be compiled. The default directory is []. The default file type is ADA. The maximum length of lines in file-spec is 255. The maximum number of source lines in file-spec is 65534. Wildcards are allowed.

Note: If you specify a wildcard the order of the compilation is alphabetical, which is not always successful. Thus wildcards should be used together with /ANALYZE_DEPENDENCY. With this qualifier the sources can be processed in any order.

Description

The source file may contain a sequence of compilation units (cf. LRM §10.1). All compilation units in the source file are compiled individually. When a compilation unit is compiled successfully, the program library is updated and the Compiler continues with the compilation of the next unit on the source file. If the compilation unit contained errors, they are reported (see §4.4). In this case, no update operation is performed on the program library and all subsequent compilation units in the compilation are only analyzed without generating code.

The Compiler delivers the status code WARNING on termination (cf. VAX/VMS, DCL Dictionary, command EXIT) if one of the compilation units contained errors. A message corresponding to this code has not been defined; hence %NONAME-W-NOMSG is printed upon notification of a batch job terminated with this status.

Command Qualifiers

`/ANALYZE_DEPENDENCY`

`/NOANALYZE_DEPENDENCY (D)`

Specifies that the Compiler only performs syntactical analysis and the analysis of the dependencies on other units. The units in file-spec are entered into the library if they are syntactically correct. The actual compilation is done later with the ADA AUTOCOMPILE command.

Note: An already existing unit with the same name as the new one is replaced and all dependent units become obsolete, unless the source file of both are identical. In this case the library is *not* updated because the dependencies are already known.

By default, the normal, full compilation is done.

/LIBRARY=dir-spec

Specifies the program library the command works on. The ADA COMPILE command needs write access to the library. The default is [.ADALIB].

/LOG[=file-spec]

/NOLOG (D)

Controls whether the Compiler writes additional messages onto the specified file. The default file name is SYS\$OUTPUT. The default file type is LOG.

By default, no additional messages are written.

/RECOMPILE

/NORECOMPILE (D)

Indicates that a recompilation of a previously analyzed source is to be performed. This qualifier should not be used unless the command was produced by the Alsys Ada Recompiler. See the ADA RECOMPILE command.

Positional Qualifiers

/AUTO_INLINE (D)

/NOAUTO_INLINE

Controls whether automatic inline expansion is performed. A subprogram S is automatically inlined at a place P where S is called, if the following conditions hold: S meets the requirements for explicit inlining via pragma INLINE (cf. §15.1.1); spec and body of S are in the same compilation unit; and the (estimated) size of the code of S is less than a fixed limit. If you specify NOAUTO_INLINE automatic inline expansion is suppressed.

By default, automatic inline expansion is performed.

/CHECK (D)

/NOCHECK

Controls whether all run-time checks are suppressed. If you specify /NOCHECK this is equivalent to the use of PRAGMA suppress for all kinds of checks.

By default, no run-time checks are suppressed, except in cases where PRAGMA suppress_all appears in the source.

/COPY_SOURCE (D)

/NOCOPY_SOURCE

Controls whether a copy of the source file is kept in the library. The copy in the program library is used for later access by the Debugger or tools like the Recompiler. The name of the copy is generated by the Compiler and need

normally not be known by the user. The Recompiler and the Debugger know this name. You can use the ADA DIRECTORY/FULL command to see the file name of the copy. If a specified file contains several compilation units a copy containing only the source text of one compilation unit is stored in the library for each compilation unit. Thus the Recompiler can recompile a single unit.

If /NOCOPY_SOURCE is specified, the Compiler only stores the name of the source file in the program library. In this case the Recompiler and the Debugger are able to use the original file if it still exists.

/COPY_SOURCE cannot be specified together with /ANALYZE_DEPENDENCY.

/INLINE (D)

/NOINLINE

Controls whether inline expansion is performed as requested by PRAGMA inline. If you specify /NOINLINE these pragmas are ignored.

By default, inline expansion is performed.

/LIST[=file-spec]

/NOLIST (D)

Controls whether a listing file is created. One listing file is created for each source file compiled. If /LIST is placed as a command qualifier a listing file is created for all sources. If /LIST is placed as a parameter qualifier a listing file is created only for the corresponding source file.

The default directory for listing files is the current default directory. The default file name is the name of the source file being compiled unless /RE-COMPILE is specified. In this case the name of the original source file, which is stored in the library, is taken as default. The default file type is LIS. No wildcard characters are allowed in the file specification.

By default, the COMPILE command does not create a listing file.

/MACHINE_CODE

/NOMACHINE_CODE (D)

Controls whether machine code is appended at the listing file. /MACHINE_CODE has no effect if /NOLIST or /ANALYZE_DEPENDENCY is specified.

By default, no machine code is appended at the listing file.

/OPTIMIZE=(FULL|PARTIAL)

/NOOPTIMIZE

Controls the level of optimization which is applied in generating code. /NOOPTIMIZE indicates no optimizations.

`/OPTIMIZE=PARTIAL` means those optimizations that do not move code globally. These are: Constant propagation, copy propagation, algebraic simplifications, runtime check elimination, dead code elimination, peephole and pipeline optimizations. This optimization level allows easier debugging while maintaining a reasonable code quality.

By default, full optimization (`/OPTIMIZE=FULL`) is done.

End of Command Description

5 Linking

The Linker of the Alsys Ada System either performs incremental linking or final linking.

Final linking produces a *program image file* which contains a loadable program. The *code portion* which is part of the program image file must be loaded onto the target later on. Final linking can (but need not) be based on the results of previous incremental linking.

Incremental linking means that a program is linked step by step (say in $N \geq 1$ steps 1 ... N). All steps except the last one are called incremental linking steps. In an incremental linking step, a *collection image file* containing a *collection* is produced; a collection is a set of Ada units and external units.

Each step $X \in \{2 \dots N\}$ is based on the result of step $X-1$. The last step is always a *final link*, i.e. it links the Ada main program. The result of an incremental linking step is also a code portion which must be loaded onto the target later on.

So the code of a program may consist of several code portions which are loaded onto the target one by one. This is called *incremental loading*.

The reasons for the introduction of the concept of incremental linking and loading into the Ada Cross System are the following:

- It should be possible that some Ada library units and external units are compiled, linked, and burnt into a ROM that is plugged into the target, and that programs using these units are linked afterwards.
- The loading time during program development should be as short as possible. This is achieved by linking those parts of the program that are not expected to be changed (e.g. some library units and the Ada Runtime System). The resulting code portion is loaded to the target and need not be linked or loaded later on. Instead, only those parts of the program that have been modified or introduced since the first link must be linked, so that the resulting code portion is much smaller in size than the code of the whole program would be. Because typically this code portion is loaded several times during program development, the development cycle time is reduced drastically.

The Runtime System (which is always necessary for the execution of Ada programs) is always linked during the first linking step. In particular, this means that also the version of the Runtime System (Debug or Non-Debug) is fixed during the first step.

The Linker gives the user great flexibility by allowing him to prescribe the mapping of single Ada units and assembler routines into the memory of the target. This, for example, enables the user to map units that are time critical into the fastest memory parts.

For this purpose the user specifies the regions of the target's memory space that are to be used by the Linker, the size of the stack, and the regions that are to be used for the stack, for the code, for the data, and for the heap. All these instructions for the Linker are contained in the *linker directive file*, which is a parameter of the ADA LINK command.

The Linker maps a set of sections into the target's memory regions. Each section belongs either to a compilation unit or to an external (assembler) unit, or is generated by the Linker itself. When mapping sections into regions, the Linker has to take into consideration the parameters and directives given by the user.

A *section* is a contiguous sequence of bytes representing code or data. A section is the smallest unit for the Linker.

5.1 Linking Main Programs

Linking a main program is called final linking. The Linker determines the compilation units belonging to the Ada program, automatically completes (see §4.2) all instances of generic units and all packages which do not require a body, determines the elaboration order and links the Ada program. Selective linking is done automatically by the Linker. Linking selectively means that only those subprograms of an imported package which are really needed are linked. This can lead to a drastic reduction of the program size, e.g. when only a few subprograms of a package providing mathematical functions are used.

Final linking results in a program image file. There is a code portion in this image file which together with the code portions of the collections belonging to the given base (if any) is the code of all Ada units and all external (assembler written) units that belong to the program and that are really needed (selective linking).

If the resulting program is to be a stand-alone program (no communication line between host and target), a small piece of the Minimal Target Kernel - the startup routine - must explicitly be linked to the program during final linking with the /KERNEL qualifier. In this case the major piece of the Minimal Target Kernel - the main part - must also be linked to the Ada program. In contrast to the startup routine you are free to give the main part either during final or during incremental linking. The main part is always given with the /EXTERNAL qualifier.

Final linking is started by the ADA LINK command. Incremental linking is started by the ADA LINK/INCREMENTAL command.

ADA LINK

Command Description

Format

| | |
|---------------------------------|-----------------------------|
| \$ ADA LINK unit | |
| Command Qualifiers | Defaults |
| /[NO]AUTO_INLINE | /AUTO_INLINE |
| /[NO]BASE[=file-spec] | /NOBASE |
| /[NO]CHECK | /CHECK |
| /[NO]COMPLETE | /COMPLETE |
| /[NO]DEBUG | /DEBUG |
| /DIRECTIVE=file-spec | /DIRECTIVE=TARGET_DIRECTIVE |
| /[NO]EXTERNAL[=(file-spec,...)] | /NOEXTERNAL |
| /IMAGE=file-spec | see Text |
| /[NO]INLINE | /INLINE |
| /[NO]KERNEL[=file-spec] | /NOKERNEL |
| /LIBRARY=directory-spec | /LIBRARY=[.ADALIB] |
| /[NO]LIST[=file-spec] | /NOLIST |
| /[NO]LOG[=file-spec] | /NOLOG |
| /[NO]MACHINE_CODE | /NOMACHINE_CODE |
| /[NO]MAP[=file-spec] | /NOMAP |
| /[NO]OPTIMIZE=(FULL PARTIAL) | /OPTIMIZE=FULL |

Command Parameters**unit**

Specifies the library unit which is the main program. This must be a parameterless library procedure.

Description

The ADA LINK command invokes the Alsys Ada Linker for final linking.

The Linker generates a program image file, the code portion of which can be loaded by the ADA LOAD command and executed by the ADA START command. The default file name of the program image file is the file name of the source file which contained the specified library unit. The default file type is LOD. The default directory is [].

Command Qualifiers

/AUTO_INLINE (D)

/NOAUTO_INLINE

This qualifier is passed to the implicitly invoked Completer. See the same qualifier with the ADA COMPLETE command.

/BASE=file-spec

/NOBASE (D)

The BASE qualifier specifies a collection image file (a file containing the result of a previous incremental link). Its default file type is LOD. The default directory is []. If a base is specified, then the final link is done on the base of the given file.

/CHECK (D)**/NOCHECK**

This qualifier is passed to the implicitly invoked Completer. See the same qualifier with the ADA COMPLETE command.

/COMPLETE (D)**/NOCOMPLETE**

Controls whether the Completer of the Alsys Ada System is invoked before the linking is performed. Only specify /NOCOMPLETE if you are sure that there are no instantiations or implicit package bodies to be compiled, e.g. if you repeat the ADA LINK command with different linker options.

/DEBUG (D)**/NODEBUG**

Controls whether debug information for the Alsys Ada Debugger is to be generated and included in the program image file. If the program is to run under the control of the Debugger it must be linked with the /DEBUG qualifier.

By default, debug information is included in the program image file.

Important: If a base is given, the resulting program is debuggable only when debug information is generated (qualifier /DEBUG) and the initial base includes the Debug version of the Runtime System (cf. 5.2).

/DIRECTIVE=file-spec**/DIRECTIVE=TARGET_DIRECTIVE (D)**

Specifies the name of the file which contains the linker directives. They describe the target's memory regions and prescribe the mapping of code, data, stack, and heap sections into these regions. The default file type is LID. The default directory is []. For its format see §5.3. The Alsys Ada System is delivered with two directive files for the IDT 7RS301. Each of them can be used for those applications that use the whole memory of the IDT 7RS301 and do not require specific units to be linked into specific regions. The difference between them is that the file <ada>:IDT_MIN must be used when the main part of Minimal Target Kernel is given with the /EXTERNAL qualifier and the file <ada>:IDT_FULL when it is not given. If you want to use one of these file, specify:

/DIRECTIVE=<ada>:IDT_FULL

or

`/DIRECTIVE=<ada>:IDT_MIN`

, where `<ada>` stands for the directory specification where the Alsys Ada System is located on your computer. This directory can be found using the `ADA SHOW LIBRARY` command. (If you have no library just create one with the `ADA CREATE` command.) The output of the `ADA SHOW LIBRARY` command displays the directory in the first line.

If this qualifier is not specified, a logical name `TARGET_DIRECTIVE` must be defined denoting the directive file to be used.

`/EXTERNAL=(file-spec,...)`

`/NOEXTERNAL (D)`

Specifies files which contain the object code of those program units which are written in assembler; these files have to be generated by the original Assembler on any MIPS/RISCos machine and then have to be transported to the host, cf. §4.5. If several files are given, they must be separated by commas. Their default directory is `[]`, their default file type is `O`.

`/IMAGE=file-spec`

Specifies the name of the file which will contain the result of the final link. The default file name of the program image is the file name of the source file which contained the specified library unit. The default file type is `LOD`. The default directory is `[]`.

`/INLINE (D)`

`/NOINLINE`

This qualifier is passed to the implicitly invoked Completer. See the same qualifier with the `ADA COMPLETE` command.

`/KERNEL=file-spec`

`/NOKERNEL (D)`

Specifies the name of the file that contains the assembled code of the Target Kernel that is to be linked to the program. The default file type is `O`. The default directory is `[]`. If you want to work with the Minimal Target Kernel that is delivered with the Alsys Ada System for the IDT 7RS301, specify:

`/KERNEL=<ada>:MTKENTRY.O,`

where `<ada>` stands for the directory specification where the Alsys Ada System is located on your computer (see the `/DIRECTIVE` qualifier on how to find that directory). This causes the startup routine of the Minimal Target Kernel to be linked to the program. The main part of the Minimal Target Kernel for IDT 7RS301 is contained in the object file `<ada>:MTKMAIN.O`. It must be given with the `/EXTERNAL` qualifier.

If the `/KERNEL` qualifier is not specified, then no Target Kernel is linked to the program. This is recommended if the Full Target Kernel is already on the target. Note, the startup routine of the Minimal Target Kernel must not be linked to the final program if the Debug Runtime System is used.

/LIBRARY=directory-spec

Specifies the program library the command works on. The ADA LINK command needs write access to the library unless /NOCOMPLETE is specified. If /NOCOMPLETE is specified the ADA LINK command needs only read access. The default library is [.ADALIB].

/LIST[=file-spec]

/NOLIST (D)

This qualifier is passed to the implicitly invoked Completer. See the same qualifier with the ADA COMPLETE command. By default the Completer does not create a listing file.

/LOG[=file-spec]

/NOLOG (D)

Controls whether the implicitly invoked Completer writes additional messages onto the specified file. The default file name is SYSS\$OUTPUT. The default file type is LOG.

By default, no additional messages are written.

/MACHINE_CODE

/NOMACHINE_CODE (D)

This qualifier is passed to the implicitly invoked Completer. See the same qualifier with the ADA COMPLETE command. If /LIST and /MACHINE_CODE is specified, the Linker of the Alsys Ada System generates a listing with the machine code of the program starter in the file LINK.LIS. The program starter is a routine which contains the calls of the necessary elaboration routines and a call for the Ada subprogram which is the main program.

By default, no machine code is generated.

/MAP[=file-spec]

/NOMAP (D)

Specifies whether the map listing of the Linker and the table of symbols which are used for linking the Ada units are to be produced in the specified file. The default directory is the directory in which the program image file is located. The default file name is the name of the program image file. The default file type is MAP.

/OPTIMIZE=(FULL|PARTIAL)

/NOOPTIMIZE

This qualifier is passed to the implicitly invoked Completer. See the same qualifier with the ADA COMPLETE command.

End of Command Description

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are contained in the following Predefined Language Environment (chapter 13 of the compiler user manual).

13 Predefined Language Environment

The predefined language environment comprises the package standard, the language-defined library units and the implementation-defined library units.

13.1 The Package STANDARD

The specification of the package standard is outlined here; it contains all predefined identifiers of the implementation.

PACKAGE standard IS

```

TYPE boolean IS (false, true);
-- The predefined relational operators for this type are as follows:
-- FUNCTION "=" (left, right : boolean) RETURN boolean;
-- FUNCTION "/=" (left, right : boolean) RETURN boolean;
-- FUNCTION "<" (left, right : boolean) RETURN boolean;
-- FUNCTION "<=" (left, right : boolean) RETURN boolean;
-- FUNCTION ">" (left, right : boolean) RETURN boolean;
-- FUNCTION ">=" (left, right : boolean) RETURN boolean;
-- The predefined logical operators and the predefined logical
-- negation operator are as follows:
-- FUNCTION "AND" (left, right : boolean) RETURN boolean;
-- FUNCTION "OR" (left, right : boolean) RETURN boolean;
-- FUNCTION "XOR" (left, right : boolean) RETURN boolean;
-- FUNCTION "NOT" (right : boolean) RETURN boolean;
-- The universal type universal_integer is predefined.
TYPE integer IS RANGE - 2_147_483_648 .. 2_147_483_647;
-- The predefined operators for this type are as follows:
-- FUNCTION "=" (left, right : integer) RETURN boolean;
-- FUNCTION "/=" (left, right : integer) RETURN boolean;
-- FUNCTION "<" (left, right : integer) RETURN boolean;
-- FUNCTION "<=" (left, right : integer) RETURN boolean;
-- FUNCTION ">" (left, right : integer) RETURN boolean;
-- FUNCTION ">=" (left, right : integer) RETURN boolean;
-- FUNCTION "+" (right : integer) RETURN integer;
-- FUNCTION "-" (right : integer) RETURN integer;
-- FUNCTION "ABS" (right : integer) RETURN integer;
-- FUNCTION "+" (left, right : integer) RETURN integer;
-- FUNCTION "-" (left, right : integer) RETURN integer;
-- FUNCTION "*" (left, right : integer) RETURN integer;
```

```

-- FUNCTION "/" (left, right : integer) RETURN integer;
-- FUNCTION "REM" (left, right : integer) RETURN integer;
-- FUNCTION "MOD" (left, right : integer) RETURN integer;
-- FUNCTION "***" (left : integer; right : integer) RETURN integer;
-- An implementation may provide additional predefined integer types.
-- It is recommended that the names of such additional types end
-- with INTEGER as in SHORT_INTEGER or LONG_INTEGER. The
-- specification of each operator for the type universal_integer, or
-- for any additional predefined integer type, is obtained by
-- replacing INTEGER by the name of the type in the specification
-- of the corresponding operator of the type INTEGER, except for the
-- right operand of the exponentiating operator.
TYPE short_short_integer IS RANGE - 128 .. 127;
TYPE short_integer IS RANGE - 32_768 .. 32_767;
-- The universal type universal_real is predefined.
TYPE float IS DIGITS 6 RANGE
    - 16#0.FFFF_F8#E32 .. 16#0.FFFF_F8#E32;
-- The predefined operators for this type are as follows:
-- FUNCTION "=" (left, right : float) RETURN boolean;
-- FUNCTION "/=" (left, right : float) RETURN boolean;
-- FUNCTION "<" (left, right : float) RETURN boolean;
-- FUNCTION "<=" (left, right : float) RETURN boolean;
-- FUNCTION ">" (left, right : float) RETURN boolean;
-- FUNCTION ">=" (left, right : float) RETURN boolean;
-- FUNCTION "+" (right : float) RETURN float;
-- FUNCTION "-" (right : float) RETURN float;
-- FUNCTION "ABS" (right : float) RETURN float;
-- FUNCTION "+" (left, right : float) RETURN float;
-- FUNCTION "-" (left, right : float) RETURN float;
-- FUNCTION "*" (left, right : float) RETURN float;
-- FUNCTION "/" (left, right : float) RETURN float;
-- FUNCTION "***" (left : float; right : integer) RETURN float;
-- An implementation may provide additional predefined floating
-- point types. It is recommended that the names of such additional
-- types end with FLOAT as in SHORT_FLOAT or LONG_FLOAT.
-- The specification of each operator for the type universal_real,
-- or for any additional predefined floating point type, is obtained
-- by replacing FLOAT by the name of the type in the specification of
-- the corresponding operator of the type FLOAT.
TYPE long_float IS DIGITS 15 RANGE
    - 16#0.FFFF_FFFF_FFFF_E#E256 .. 16#0.FFFF_FFFF_FFFF_E#E256;
-- In addition, the following operators are predefined for universal
-- types:
-- FUNCTION "*" (left : UNIVERSAL_INTEGER; right : UNIVERSAL_REAL)
--     RETURN UNIVERSAL_REAL;
-- FUNCTION "*" (left : UNIVERSAL_REAL; right : UNIVERSAL_INTEGER)
--     RETURN UNIVERSAL_REAL;

```

```
-- FUNCTION "/" (left : UNIVERSAL_REAL;    right : UNIVERSAL_INTEGER)
      RETURN UNIVERSAL_REAL;
-- The type universal_fixed is predefined.
-- The only operators declared for this type are
-- FUNCTION "*" (left : ANY_FIXED_POINT_TYPE;
      right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;
-- FUNCTION "/" (left : ANY_FIXED_POINT_TYPE;
      right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;
-- The following characters form the standard ASCII character set.
-- Character literals corresponding to control characters are not
-- identifiers.
```

```
TYPE character IS
```

```
(nul,  soh,  stx,  etx,      eot,  enq,  ack,  bel,
  bs,   ht,   lf,   vt,      fr,   cr,   so,   si,
  dle,  dc1,  dc2,  dc3,      dc4,  nak,  syn,  etb,
  can,  em,   sub,  esc,      fs,   gs,   rs,   us,
  ' ',  '!',  '"',  '#',      '$',  '%',  '&',  '...',
  '(',  ')',  '*',  '+',      ',',  '-',  '.',  '/',
  '0',  '1',  '2',  '3',      '4',  '5',  '6',  '7',
  '8',  '9',  ':',  ';',      '<',  '=',  '>',  '?',
  '@',  'A',  'B',  'C',      'D',  'E',  'F',  'G',
  'H',  'I',  'J',  'K',      'L',  'M',  'N',  'O',
  'P',  'Q',  'R',  'S',      'T',  'U',  'V',  'W',
  'X',  'Y',  'Z',  '[',      '\',  ']',  '^',  '_',
  '...', 'a',  'b',  'c',      'd',  'e',  'f',  'g',
  'h',  'i',  'j',  'k',      'l',  'm',  'n',  'o',
  'p',  'q',  'r',  's',      't',  'u',  'v',  'w',
  'x',  'y',  'z',  '{',      '|',  '}',  '~',  del);
```

```
FOR character USE -- 128 ascii CHARACTER SET WITHOUT HOLES
      (0, 1, 2, 3, 4, 5, ..., 125, 126, 127);
```

```
-- The predefined operators for the type CHARACTER are the same as
-- for any enumeration type.
```

```
PACKAGE ascii IS
```

```
-- Control characters:
```

```
nul : CONSTANT character := nul;    soh : CONSTANT character := soh;
stx : CONSTANT character := stx;    etx : CONSTANT character := etx;
eot : CONSTANT character := eot;    enq : CONSTANT character := enq;
ack : CONSTANT character := ack;    bel : CONSTANT character := bel;
bs  : CONSTANT character := bs;     ht  : CONSTANT character := ht;
lf  : CONSTANT character := lf;     vt  : CONSTANT character := vt;
ff  : CONSTANT character := ff;     cr  : CONSTANT character := cr;
so  : CONSTANT character := so;     si  : CONSTANT character := si;
dle : CONSTANT character := dle;    dc1 : CONSTANT character := dc1;
dc2 : CONSTANT character := dc2;    dc3 : CONSTANT character := dc3;
dc4 : CONSTANT character := dc4;    nak : CONSTANT character := nak;
syn : CONSTANT character := syn;    etb : CONSTANT character := etb;
can : CONSTANT character := can;    em  : CONSTANT character := em;
```

```

sub : CONSTANT character := sub;   esc : CONSTANT character := esc;
fs  : CONSTANT character := fs;    gs  : CONSTANT character := gs;
rs  : CONSTANT character := rs;    us  : CONSTANT character := us;
del : CONSTANT character := del;

-- Other characters:
exclam    : CONSTANT character := '!';
quotation : CONSTANT character := '"';
sharp     : CONSTANT character := '#';
dollar    : CONSTANT character := '$';
percent   : CONSTANT character := '%';
ampersand : CONSTANT character := '&';
colon     : CONSTANT character := ':';
semicolon : CONSTANT character := ';';
query     : CONSTANT character := '?';
at_sign   : CONSTANT character := '@';
l_bracket : CONSTANT character := '[';
back_slash : CONSTANT character := '\';
r_bracket : CONSTANT character := ']';
circumflex : CONSTANT character := '^';
underline : CONSTANT character := '_';
grave     : CONSTANT character := '`';
l_brace   : CONSTANT character := '{';
bar       : CONSTANT character := '|';
r_brace   : CONSTANT character := '}';
tilde     : CONSTANT character := '~';
lc_a      : CONSTANT character := 'a';
...
lc_z      : CONSTANT character := 'z';
END ascii;

-- Predefined subtypes:
SUBTYPE natural IS integer RANGE 0 .. integer'last;
SUBTYPE positive IS integer RANGE 1 .. integer'last;

-- Predefined string type:
TYPE string IS ARRAY(positive RANGE <>) OF character;
PRAGMA pack(string);

-- The predefined operators for this type are as follows:
-- FUNCTION "=" (left, right : string) RETURN boolean;
-- FUNCTION "/=" (left, right : string) RETURN boolean;
-- FUNCTION "<" (left, right : string) RETURN boolean;
-- FUNCTION "<=" (left, right : string) RETURN boolean;
-- FUNCTION ">" (left, right : string) RETURN boolean;
-- FUNCTION ">=" (left, right : string) RETURN boolean;
-- FUNCTION "&" (left : string; right : string) RETURN string;
-- FUNCTION "&" (left : character; right : string) RETURN string;
-- FUNCTION "&" (left : string; right : character) RETURN string;
-- FUNCTION "&" (left : character; right : character) RETURN string;
TYPE duration IS DELTA 2#1.0#E-14 RANGE

```



```
    - 131_072.0 .. 131_071.999_938_964_843_75;
-- The predefined operators for the type DURATION are the same
-- as for any fixed point type.
-- the predefined exceptions:
constraint_error : EXCEPTION;
numeric_error    : EXCEPTION;
program_error    : EXCEPTION;
storage_error    : EXCEPTION;
tasking_error    : EXCEPTION;
END standard;
```

13.2 Language-Defined Library Units

The following language-defined library units are included in the master library:

- The package system
- The package calendar
- The generic procedure unchecked_deallocation
- The generic function unchecked_conversion
- The package io_exceptions
- The generic package sequential_io
- The generic package direct_io
- The package text_io
- The package low_level_io

13.3 Implementation-Defined Library Units

The master library also contains the implementation-defined library units

- collection_manager,
- timing, and
- privileged_operations.

15 Appendix F

This chapter, together with the Chapters 16 and 17, is the Appendix F required in the LRM, in which all implementation-dependent characteristics of an Ada implementation are described.

15.1 Implementation-Dependent Pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

15.1.1 Predefined Language Pragmas

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here.

CONTROLLED
has no effect.

ELABORATE
is fully implemented. The Alsys Ada System assumes a PRAGMA elaborate, i.e. stores a unit in the library as if a PRAGMA elaborate for a unit *u* was given, if the compiled unit contains an instantiation of *u* (or a generic program unit in *u*) and if it is clear that *u* *must* have been elaborated before the compiled unit. In this case an appropriate information message is given. By this means it is avoided that an elaboration order is chosen which would lead to a PROGRAM_ERROR when elaborating the instantiation.

INLINE
Inline expansion of subprograms is supported with the following restrictions: the subprogram must not contain declarations of other subprograms, tasks, generic units or body stubs. If the subprogram is called recursively only the outer call of this subprogram will be expanded.

INTERFACE

is supported for ASSEMBLER. `PRAGMA interface(assembler, ...)` provides an interface with the internal calling conventions of the Alsys Ada System. See §15.1.3 for further description.

`PRAGMA interface` should always be used in connection with the `PRAGMA external_name` (see §15.1.2), otherwise the Compiler will generate an internal name that leads to an unsolved reference during linking. These generated names are prefixed with an underline; therefore the user should not use names beginning with an underline.

LIST

is fully implemented. Note that a listing is only generated when the `/LIST` qualifier is specified with the `ADA COMPILE` (or `ADA COMPLETE` or `ADA LINK`) command.

MEMORY_SIZE

has no effect.

OPTIMIZE

has no effect; but see also the `/OPTIMIZE` qualifier with the `ADA COMPILE` command, §4.1

PACK

see §16.1.

PAGE

is fully implemented. Note that form feed characters in the source do not cause a new page in the listing. They are - as well the other format effectors (horizontal tabulation, vertical tabulation, carriage return, and line feed) - replaced by a ~ character in the listing.

PRIORITY

There are two implementation-defined aspects of this pragma: First, the range of the subtype priority, and second, the effect on scheduling (Chapter 14) of not giving this pragma for a task or main program. The range of subtype priority is 0 .. 15, as declared in the predefined library package `system` (see §15.3); and the

effect on scheduling of leaving the priority of a task or main program undefined by not giving PRAGMA priority for it is the same as if the PRAGMA priority 0 had been given (i.e. the task has the lowest priority).

SHARED

is fully supported.

STORAGE_UNIT

has no effect.

SUPPRESS

has no effect, but see §15.1.2 for the implementation-defined PRAGMA `suppress_all`.

SYSTEM_NAME

has no effect.

15.1.2 Implementation-Defined Pragmas**BYTE_PACK**

see §16.1.

EXTERNAL_NAME (<string>, <ada_name>)

<ada_name> specifies the name of a subprogram or of an object declared in a library package, <string> must be a string literal. It defines the external name of the specified item. The Compiler uses a symbol with this name in the call instruction for the subprogram. The subprogram declaration of <ada_name> must precede this pragma. If several subprograms with the same name satisfy this requirement the pragma refers to that subprogram which is declared last.

Upper and lower cases are distinguished within <string>, i.e. <string> must be given exactly as it is to be used by external routines. This pragma will be used in connection with the pragma interface (assembler) (see §15.1.1).

RESIDENT (<ada_name>)

this pragma causes the value of the object <ada_name> to be held in memory and prevents assignments of a value to the object <ada_name> from being eliminated by the optimizer (see §4.1) of the Alsys Ada Compiler.

This pragma can be applied to all those kinds of objects for which the address clause is supported (cf. §16.5).

SUPPRESS_ALL

causes all the runtime checks described in the LRM §11.7 to be suppressed; this pragma is only allowed at the start of a compilation before the first compilation unit; it applies to the whole compilation.

15.1.3 Pragma Interface (Assembler,...)

This section describes the internal calling conventions of the Alsys Ada System, which are the same ones which are used for subprograms for which a PRAGMA interface (ASSEMBLER,...) is given. Thus the actual meaning of this pragma is simply that the body needs and must not be provided in Ada, but in object form using the /EXTERNAL qualifier with the ADA LINK command.

The internal calling conventions are explained in four steps:

- Parameter passing mechanism
- Ordering of parameters
- Type mapping
- Saving registers

Parameter passing mechanism:

The Alsys Ada System uses three different parameter passing mechanisms, depending on the type of a parameter:

- *by value and/or result*: The value of the parameter itself is passed.
- *by reference*: The address of the parameter is passed (like an IN parameter of type `system.address`, which would be passed by value).
- *by descriptor*: A descriptor for the parameter is allocated on the caller's side and is itself passed by reference.

The parameters of a subprogram are passed in registers where possible. The remaining parameters, if any, are passed in an area called a *parameter block*. This area is aligned on a word boundary and contains parameter values (for parameter of scalar types), parameter addresses or descriptor addresses (for parameter of composite types) and alignment gaps.

For a function subprogram an extra register (\$r4 or \$f0) is assigned to contain the function result upon return. Thus the return value of a function is treated like an anonymous parameter of mode OUT. No special treatment is required for a function result except for return values of an unconstrained array type (see below).

A subprogram is called using the JAL instruction. The address of the parameter block is passed in \$r3, if necessary. The static link of a subprogram is passed in \$r2, if necessary.

When determining the position of a parameter within the parameter block, the calling mechanism and the size and alignment requirements of the parameter type are considered. The size and alignment requirements and the passing mechanism are as follows:

Scalar parameters and parameters of access types are passed by value, i.e. the values of the actual parameters of modes IN or IN OUT are copied into the parameter register or into the parameter block before the call. Then, after the subprogram has returned, values of the actual parameters of modes IN OUT and OUT are copied out of the parameter register or the parameter block into the associated actual parameters. The parameters are aligned within the parameter block according their size: A parameter with a size of 8, 16 or 32 bits has an alignment of 1, 2 or 4 (which means that the object is aligned to a byte, halfword or word boundary within the parameter block). If the size of the parameter is not a multiple of 8 bits (which may be achieved by attaching a size specification to the parameter's type in case of an integer, enumeration or fixed point type) it will be byte aligned. Parameters of access types are always aligned to a word boundary.

Parameters of composite types are passed by reference or by descriptor. The descriptors are allocated by the caller and are themselves passed by reference. A descriptor contains the address of the actual parameter object and further information dependent on the specific parameter type. The following composite parameter types are distinguished:

- A parameter of a constrained array type is passed by reference for all parameter modes.
- For a parameter of an unconstrained array type, the descriptor consists of the address of the actual array parameter followed by the bounds for each index range in the array (i.e. FIRST(1), LAST(1), FIRST(2), LAST(2), ...). The space allocated for the bound elements in the descriptor depends on the type of the index constraint. This descriptor is itself passed by reference.
- For functions whose return value is an unconstrained array type, a reference to a descriptor for the array is passed in the parameter block as for parameters of

mode OUT. The fields for its address and all array index bounds are filled up by the function before it returns. In contrast to the procedure for an OUT parameter, the function allocates the array in its own stack space. The function then returns without releasing its stack space. After the function has returned, the calling routine copies the array into its own memory space and then deallocates the stack memory of the function.

- A constrained record parameter is passed by reference for all parameter modes.
- For an unconstrained record parameter of mode IN, the parameter is passed by reference using the address pointing to the record. If the parameter has mode OUT or IN OUT, the value of the CONSTRAINED attribute applied to the actual parameter is passed as an additional boolean IN parameter (which, when not passed in a register, occupies one byte in the parameter block and is aligned to a byte boundary). The boolean IN parameter and the address are treated like two consecutive parameters in a subprogram specification, i.e. the positions of the two parameters within the parameter block are determined independently of each other.

For all kinds of composite parameter types, the pointer pointing to the actual parameter object is represented by a 32 bit address, which is always aligned to a word boundary.

Ordering of parameters:

The ordering of the parameters is determined as follows:

The parameters are processed in the order they are defined in the Ada subprogram specification. For a function, the return value is treated as an anonymous parameter of mode OUT at the start of the parameter list. The registers \$r4..\$r22 and \$f0..\$f31 are available for parameter passing. A parameter block is only used when there are more parameters than registers of the appropriate class. Registers are used from low numbers to high numbers, the parameter block starts at offset zero and grows to higher offsets. Each parameter is handled as follows:

- A float parameter is allocated the next free even numbered floating point register (the corresponding odd numbered floating point register is not used for parameter passing). If no floating point register is available any more, one word is allocated in the parameter block (see below).
- A long_float parameter is allocated the next free floating point register pair. If no floating point register is available any more, a double word is allocated in the parameter block (see below).
- All other parameters (or their [descriptor] addresses, respectively) are allocated the next free general purpose register from \$r4..\$r22. If not enough general purpose registers are available for parameter passing, space is allocated in the parameter block depending on the representation of the parameter type (see below).

- If a parameter cannot be passed in a register, space is allocated in the parameter block as follows:

Because of the size and alignment requirements of a parameter it is not always possible to place parameters in such a way that two consecutive parameters are densely located in the parameter block. In such a situation a gap, i.e. a piece of memory space which is not associated with a parameter, exists between two adjacent parameters. Consequently, the size of the parameter block will be larger than the sum of the sizes of all the parameters.

In order to minimize the size of the gaps in a parameter block, an attempt is made to fill each gap with a parameter that occurs later in the parameter list. If during the allocation of space within the parameter block a parameter is encountered whose size and alignment fit the characteristics of an available gap, then this gap is allocated for the parameter instead of appending it at the end of the parameter block. As each parameter will be aligned to a byte, halfword or word boundary the size of any gap may be one, two or three bytes. Every gap of size three bytes can be treated as two gaps, one of size one byte with an alignment of 1 and one of size two bytes with an alignment of 2. So if a parameter of size two is to be allocated, a two byte gap, if available, is filled up. A parameter of size one will fill a one byte gap. If none exists but a two byte gap is available, this is used as two one byte gaps. By this first fit algorithm all parameters are processed in the order they occur in the Ada program.

A called subprogram accesses each parameter for reading or writing using the parameter register or using the parameter block address incremented by an offset from the start of the parameter block suitable for the parameter. So the value of a parameter of a scalar type or an access type is read (or written) directly from (into) the parameter register or parameter block. For a parameter of a composite type passed by reference the actual parameter value is accessed indirectly via the parameter address passed in a parameter register or in the parameter block. For a parameter of a composite type passed by descriptor the actual parameter value is accessed via the descriptor whose address is passed in a parameter register or in the parameter block. The descriptor contains a pointer to the actual object. When standard entry code sequences are used within the assembler subprogram (see below), the parameter block address is accessible at address -12(\$r30).

Type mapping:

To access individual components of array or record types, knowledge about the type mapping for array and record types is required. An array is stored as a sequential concatenation of all its components. Normally, pad bits are used to fill each component to a byte, word, longword or a multiple thereof depending on the size and alignment requirements of the components' subtype. This padding may be influenced using one of the PRAGMAs `pack` or `byte_pack` (cf. §16.1). The offset of an individual array

component is then obtained by multiplying the padded size of one array component by the number of components stored in the array before it. This number may be determined from the number of elements for each dimension using the fact that the array elements are stored row by row. (For unconstrained arrays the number of elements for each dimension can be found in the descriptor, which is passed by reference.)

A record object is implemented as a concatenation of its components. Initially, locations are reserved for those components that have a component clause applied to them. Then locations for all other components are reserved. Any gaps large enough to hold components without component clauses are filled, so in general the record components are rearranged. Components in record variants are overlaid. The ordering mechanism of the components within a record is in principle the same as that for ordering the parameters in the parameter block.

A record may hold implementation-dependent components (cf. §16.4). For a record component whose size depends on discriminants, a generated component holds the offset of the record component within the record object. If a record type includes variant parts there may be a generated component (cf. §16.4) holding the size of the record object. This size component is allocated as the first component within the record object if this location is not reserved by a component clause. Since the mapping of record types is rather complex record component clauses should be introduced for each record component if an object of that type is to be passed to a non Ada subprogram to be sure to access the components correctly.

Saving registers:

The last aspect of the calling conventions discussed here is that of saving registers. The calling subprogram assumes that the values of the registers \$r1..\$r22, \$r24..\$r25 will be destroyed by the called subprogram, and therefore saves them of its own accord. The stack pointer \$r29 will have the same value after the call as before except for functions returning unconstrained arrays. The stack limit register (\$r23) will have the same value after the call as before unless the stack of the main task was extended. If the called subprogram wants to modify further registers it has to ensure that the old values are restored upon return from the subprogram. Note that these register saving conventions differ from the C calling standard.

Finally we give the appropriate code sequences for the subprogram entry and for the return, which both obey the rules stated above.

A subprogram for which PRAGMA interface (assembler,...) is specified is - in effect - called with the subprogram calling sequence

```

move    $4,...      | assign IN parameters, if any
move    $f0,...
...

```

```

jal    <subprogram address>
...
move   ....,$4      | read OUT parameters, if any
...

```

Thus the appropriate entry code sequence is

```

addiu   $sp,$sp,-12
sw      $0,0($sp)
sw      $fp,4($sp)
sw      $31,8($sp)
addiu   $15,$sp,-<frame_size-4>
addiu   $fp,$sp,4
slt     $1,$23,$15
bne     $1,$0,L1
nop
jal     _EXCRSST      | raise storage error
nop
L1:
move    $sp,$15
        | The field at address -4($fp) is reserved
        | for use by the Ada runtime system

```

The return code sequence is then

```

move    $sp,$fp
lw      $31,4($sp)
lw      $fp,0($sp)
jr      $31
addiu   $sp,$sp,8

```

15.2 Implementation-Dependent Attributes

The name, type and implementation-dependent aspects of every implementation-dependent attribute is stated in this section.

15.2.1 Language-Defined Attributes

The name and type of all the language-defined attributes are as given in the LRM. We note here only the implementation-dependent aspects.

ADDRESS

If this attribute is applied to an object for which storage is allocated, it yields the address of the first storage unit that is occupied by the object.

If it is applied to a subprogram or to a task, it yields the address of the entry point of the subprogram or task body.

If it is applied to a task entry for which an address clause is given, it yields the address given in the address clause.

For any other entity this attribute is not supported and will return the value `system.address_zero`.

IMAGE

The image of a character other than a graphic character (cf. LRM §3.5.5(11)) is the string obtained by replacing each italic character in the indication of the character literal (given in the LRM Annex C(13)) by the corresponding upper-case character. For example, `character'image(nul) = "NUL"`.

MACHINE_OVERFLOW

Yields true for each real type or subtype.

MACHINE_ROUND

Yields true for each real type or subtype.

STORAGE_SIZE

The value delivered by this attribute applied to an access type is as follows:

If a length specification (`STORAGE_SIZE`, see §16.2) has been given for that type (static collection), the attribute delivers that specified value.

In case of a dynamic collection, i.e. no length specification by `STORAGE_SIZE` given for the access type, the attribute delivers the number of storage units currently allocated for the collection. Note that dynamic collections are extended if needed.

If the collection manager (cf. §13.3.1) is used for a dynamic collection the attribute delivers the number of storage units currently allocated for the collection. Note that in this case the number of storage units currently allocated may be decreased by release operations.

The value delivered by this attribute applied to a task type or task object is as follows:

If a length specification (`STORAGE_SIZE`, see §16.2) has been given for the task type, the attribute delivers that specified value; otherwise, the default value is returned.

15.2.2 Implementation-Defined Attributes

There are no implementation-defined attributes.

15.3 Specification of the Package SYSTEM

The package system as required in the LRM §13.7 is reprinted here with all implementation-dependent characteristics and extensions filled in.

PACKAGE system IS

TYPE designated_by_address IS LIMITED PRIVATE;

TYPE address IS ACCESS designated_by_address;
FOR address's storage_size USE 0;

address_zero : CONSTANT address := NULL;

FUNCTION "+" (left : address; right : integer) RETURN address;

FUNCTION "+ " (left : integer; right : address) RETURN address;

FUNCTION "-" (left : address; right : integer) RETURN address;

FUNCTION "- " (left : address; right : address) RETURN integer;

SUBTYPE external_address IS STRING;

-- External addresses use hexadecimal notation with characters

-- '0'..'9', 'a'..'f' and 'A'..'F'. For instance:

-- "7FFFFFFF"

-- "80000000"

-- "8" represents the same address as "00000008"

FUNCTION convert_address (addr : external_address) RETURN address;

-- convert_address raises CONSTRAINT_ERROR if the external address

```

-- addr is the empty string, contains characters other than
-- '0'..'9', 'a'..'f', 'A'..'F' or if the resulting address value
-- cannot be represented with 32 bits.

FUNCTION convert_address (addr : address) RETURN external_address;
-- The resulting external address consists of exactly 8 characters
-- '0'..'9', 'A'..'F'.

TYPE name IS (mips_bare);
system_name : CONSTANT name := mips_bare;

storage_unit : CONSTANT := 8;
memory_size : CONSTANT := 2 ** 31;
min_int      : CONSTANT := - 2 ** 31;
max_int      : CONSTANT := 2 ** 31 - 1;
max_digits   : CONSTANT := 15;
max_mantissa : CONSTANT := 31;
fine_delta   : CONSTANT := 2.0 ** (-31);
tick         : CONSTANT := 2.0 ** (-14);

SUBTYPE priority IS integer RANGE 0 .. 15;

TYPE interrupt_number IS RANGE 1 .. 32;
-- User defined interrupts

interrupt_vector : ARRAY (interrupt_number) OF address;
-- converts an interrupt_number to an address;

non_ada_error : EXCEPTION RENAMES _non_ada_error;
-- non_ada_error is raised, if some event occurs which does not
-- correspond to any situation covered by Ada, e.g.:
--   illegal instruction encountered
--   error during address translation
--   illegal address

TYPE exception_id IS NEW address;

no_exception_id : CONSTANT exception_id := NULL;

-- Coding of the predefined exceptions:

FUNCTION constraint_error_id RETURN exception_id;
FUNCTION numeric_error_id   RETURN exception_id;
FUNCTION program_error_id   RETURN exception_id;
FUNCTION storage_error_id   RETURN exception_id;
FUNCTION tasking_error_id   RETURN exception_id;

```

```

FUNCTION non_ada_error_id    RETURN exception_id;
FUNCTION status_error_id    RETURN exception_id;
FUNCTION mode_error_id      RETURN exception_id;
FUNCTION name_error_id      RETURN exception_id;
FUNCTION use_error_id       RETURN exception_id;
FUNCTION device_error_id    RETURN exception_id;
FUNCTION end_error_id       RETURN exception_id;
FUNCTION data_error_id      RETURN exception_id;
FUNCTION layout_error_id    RETURN exception_id;
FUNCTION time_error_id      RETURN exception_id;

```

```

no_error_code      : CONSTANT := 0;

```

```

TYPE exception_information
  IS RECORD
    excp_id          : exception_id;
    -- Identification of the exception. The codings of
    -- the predefined exceptions are given above.
    code_addr        : address;
    -- Code address where the exception occurred. Depending
    -- on the kind of the exception it may be be address of
    -- the instruction which caused the exception, or it
    -- may be the address of the instruction which would
    -- have been executed if the exception had not occurred.
    error_code        : integer;
  END RECORD;

```

```

PROCEDURE get_exception_information
  (excp_info : OUT exception_information);
-- The subprogram get_exception_information must only be called
-- from within an exception handler BEFORE ANY OTHER EXCEPTION
-- IS RAISED. It then returns the information record about the
-- actually handled exception.
-- Otherwise, its result is undefined.

```

```

PROCEDURE raise_exception_id
  (excp_id : exception_id);

```

```

PROCEDURE raise_exception_info
  (excp_info : exception_information);
-- The subprogram raise_exception_id raises the exception
-- given as parameter. It corresponds to the RAISE statement.

-- The subprogram raise_exception_info raises the exception
-- described by the information record supplied as parameter.
-- In addition to the subprogram raise_exception_id it allows to
-- explicitly define all components of the exception information record.

```

```
-- IT IS INTENDED THAT BOTH SUBPROGRAMS ARE USED ONLY WHEN
-- INTERFACING WITH THE OPERATING SYSTEM.
```

```
TYPE exit_code IS NEW integer;
```

```
error      : CONSTANT exit_code := 1;
success    : CONSTANT exit_code := 0;
```

```
PROCEDURE set_exit_code (val : exit_code);
  -- Specifies the exit code which is returned to the
  -- operating system if the Ada program terminates normally.
  -- The default exit code is 'success'. If the program is
  -- abandoned because of an exception, the exit code is
  -- 'error'.
```

```
PRIVATE
```

```
  -- private declarations
```

```
END system;
```

15.4 Restrictions on Representation Clauses

See Chapter 16 of this manual.

15.5 Conventions for Implementation-Generated Names

There are implementation generated components but these have no names. (cf. §16.4 of this manual).

15.6 Expressions in Address Clauses

See §16.5 of this manual.

15.7 Restrictions on Unchecked Conversions

The implementation supports unchecked type conversions for all kinds of source and target types with the restriction that the target type must not be an unconstrained array type. The result value of the unchecked conversion is unpredictable, if

`target_type'SIZE > source_type'SIZE`

15.8 Characteristics of the Input-Output Packages

The implementation-dependent characteristics of the input-output packages as defined in the LRM Chapter 14 are reported in Chapter 17 of this manual.

15.9 Requirements for a Main Program

A main program must be a parameterless library procedure. This procedure may be a generic instantiation; the generic procedure need not be a library unit.

15.10 Unchecked Storage Deallocation

The generic procedure `unchecked_deallocation` is provided; the effect of calling an instance of this procedure is as described in the LRM §13.10.1.

The implementation also provides an implementation-defined package `collection_manager`, which has advantages over unchecked deallocation in some applications (cf. §13.3.1).

Unchecked deallocation and operations of the `collection_manager` can be combined as follows:

- `collection_manager.reset` can be applied to a collection on which unchecked deallocation has also been used. The effect is that storage of all objects of the collection is reclaimed.
- After the first `unchecked_deallocation (release)` on a collection, all following calls of `release (unchecked deallocation)` until the next `reset` have no effect, i.e. storage is not reclaimed.

- after a `reset` a collection can be managed by `mark` and `release` (resp. `unchecked_deallocation`) with the normal effect even if it was managed by `unchecked_deallocation` (resp. `mark` and `release`) before the `reset`.

15.11 Machine Code Insertions

A package `machine_code` is not provided and machine code insertions are not supported.

15.12 Numeric Error

The predefined exception `numeric_error` is never raised implicitly by any predefined operation; instead the predefined exception `constraint_error` is raised.

16 Appendix F: Representation Clauses

In this chapter we follow the section numbering of Chapter 13 of the LRM and provide notes for the use of the features described in each section.

16.1 Pragmas

PACK

As stipulated in the LRM §13.1, this pragma may be given for a record or array type. It causes the Compiler to select a representation for this type such that gaps between the storage areas allocated to consecutive components are minimized. For components whose type is an array or record type the PRAGMA PACK has no effect on the mapping of the component type. For all other component types the Compiler will choose a representation for the component type that needs minimal storage space (packing down to the bit level). Thus the components of a packed data structure will in general not start at storage unit boundaries.

BYTE_PACK

This is an implementation-defined pragma which takes the same argument as the predefined language PRAGMA PACK and is allowed at the same positions. For components whose type is an array or record type the PRAGMA BYTE_PACK has no effect on the mapping of the component type. For all other component types the Compiler will try to choose a more compact representation for the component type. But in contrast to PRAGMA PACK all components of a packed data structure will start at storage unit boundaries and the size of the components will be a multiple of `system.storage_unit`. Thus, the PRAGMA BYTE_PACK does not effect packing down to the bit level (for this see PRAGMA PACK).

16.2 Length Clauses

SIZE

For all integer, fixed point and enumeration types the value must be ≤ 32 ;
for float types the value must be ≤ 32 (this is the amount of storage which is associated with these types anyway);
for long_float types the value must be ≤ 64 (this is the amount of storage which is associated with these types anyway).
for access types the value must be ≤ 32 (this is the amount of storage which is associated with these types anyway).
If any of the above restrictions are violated, the Compiler responds with a RESTRICTION error message in the Compiler listing.

STORAGE_SIZE

Collection size: If no length clause is given, the storage space needed to contain objects designated by values of the access type and by values of other types derived from it is extended dynamically at runtime as needed. If, on the other hand, a length clause is given, the number of storage units stipulated in the length clause is reserved, and no dynamic extension at runtime occurs.

Storage for tasks: The memory space reserved for a task is 10K bytes if no length clause is given (cf. Chapter 14). If the task is to be allotted either more or less space, a length clause must be given for its task type, and then all tasks of this type will be allotted the amount of space stipulated in the length clause (the activation of a small task requires about 1.4K bytes). Whether a length clause is given or not, the space allotted is not extended dynamically at runtime.

SMALL

There is no implementation-dependent restriction. Any specification for SMALL that is allowed by the LRM can be given. In particular those values for SMALL are also supported which are not a power of two.

16.3 Enumeration Representation Clauses

The integer codes specified for the enumeration type have to lie inside the range of the largest integer type which is supported; this is the type integer defined in package standard.

16.4 Record Representation Clauses

Record representation clauses are supported. The value of the expression given in an alignment clause must be 0, 1, 2 or 4. If this restriction is violated, the Compiler responds with a **RESTRICTION** error message in the Compiler listing. If the value is 0 the objects of the corresponding record type will not be aligned, if it is 1, 2 or 4 the starting address of an object will be a multiple of the specified alignment.

The number of bits specified by the range of a component clause must not be greater than the amount of storage occupied by this component. (Gaps between components can be forced by leaving some bits unused but not by specifying a bigger range than needed.) Violation of this restriction will produce a **RESTRICTION** error message.

There are implementation-dependent components of record types generated in the following cases :

- If the record type includes variant parts and the difference between the sizes of the maximum and the minimum variant is greater than 32 bytes, and, in addition, if it has either more than one discriminant or else the only discriminant may hold more than 256 different values, the generated component holds the size of the record object. (If the second condition is not fulfilled, the number of bits allocated for any object of the record type will be the value delivered by the size attribute applied to the record type.)
- If the record type includes array or record components whose sizes depend on discriminants, the generated components hold the offsets of these record components (relative to the corresponding generated component) in the record object.

But there are no implementation-generated names (cf. LRM §13.4(8)) denoting these components. So the mapping of these components cannot be influenced by a representation clause.

16.5 Address Clauses

Address clauses are supported for objects declared by an object declaration and for single task entries. If an address clause is given for a subprogram, package or a task unit, the Compiler responds with a **RESTRICTION** error message in the Compiler listing.

If an address clause is given for an object, the storage occupied by the object starts at the given address. Address clauses for single entries are described in §16.5.1.

16.5.1 Interrupts

If an address clause is given for a task entry, then this entry can be called by a user written interrupt service routine. Such an entry is called an interrupt entry. While an interrupt entry is identified by its Ada name within the Ada program, it is identified by a number (`TYPE interrupt_number of PACKAGE system`) outside the Ada program (e.g. in the interrupt service routine). This number is defined by the address clause for the interrupt entry, as the following example shows, where an interrupt entry `intr_entry` is declared with number 1:

```
ENTRY intr_entry;
FOR intr_entry USE AT system.interrupt_vector (1);
```

The number of an interrupt entry allows an interrupt service routine to call the interrupt entry. Parameters for interrupt entries are not supported.

The connection between a hardware interrupt and an interrupt entry is done within an interrupt service routine. An interrupt service routine is an assembler routine that is automatically activated each time a specific hardware interrupt occurs.

Suppose, for example, that you want to catch the hardware interrupt `INTR` (`INTR` represents the number (0 through 5) of the hardware interrupt to be caught). Then you must write an assembler routine, called `ISR` in the following, which will be activated each time the interrupt `INTR` occurs. This routine must be defined as an interrupt service routine by calling the `PROCEDURE define_interrupt_service_routine` of `PACKAGE privileged_operations`. In the Ada program, `ISR` must be declared as an interrupt service routine as follows:

```
-- Declaration of the external routine ISR:
PROCEDURE isr;
  PRAGMA interface (assembler, isr);
  PRAGMA external_name ("ISR", isr);

-- Definition of ISR as interrupt service routine for interrupt INTR:
define_interrupt_service_routine (isr'address, INTR);
```

The interrupt service routine `ISR` must look like this:

```
ISR:  ...      --- save all used registers
      ...      --- action
      ...
      ...      --- restore all used registers
      J      $31
```

NOP

It is very important that when leaving ISR all registers except \$1 have the same values as they had when entering ISR. ISR is executed in the kernel mode of the processor, so all instructions (including privileged ones) can be used within ISR. The priority of ISR (cf. §19.1.2) depends on the interrupt source.

Once ISR has returned control back to the Kernel, the reason for INTR, i.e. the corresponding bit in the Cause register must have been cleared by ISR. Otherwise ISR will immediately be invoked again which causes an endless loop.

If you want to call the interrupt entry with the number N, then you must set a bit within the interrupt entry call pending indicator `_IREENTRYC` by the instructions:

```

LA    $1, _IREENTRYC
                    #-- prepare call of interrupt entry N
LW    $2, 0($1)
NOP
ORI   $2, 1<<(N-1)
SW    $2, 0($1)

```

A complete example for interrupt handling follows. For this example the second RS232 serial line of the IDT 7RS301 is used (available through the tty1 connector). The assembler routine `ISR_READ` is activated each time a character is received on that line. `ISR_READ` reads the character (this reading removes INTR, i.e. clears the IP-bit in the Cause register) and stores the character in a variable defined by a global package. `ISR_READ` then calls interrupt entry `char_entry` of TASK `terminal_in`. `terminal_in` uses TASK `terminal_out` to output each character read. The terminal should be set up for XON/XOFF (not CTS/RTS) flow control.

```
PACKAGE terminal_global IS
```

```

    the_char : character;
    PRAGMA external_name ("THE_CHAR", the_char);

```

```

END terminal_global;
WITH privileged_operations,
    system,
    terminal_global,
    text_io;

```

```

USE privileged_operations,
    terminal_global,
    text_io;

```

PROCEDURE terminal IS

PRAGMA priority (2);

PROCEDURE isr_read;

PRAGMA interface (assembler, isr_read);

PRAGMA external_name ("ISR_READ", isr_read);

kseg1 : CONSTANT address := convert_address ("A0000000");

dua_adr : CONSTANT address := kseg1 + 16#1FE00000#;

dua_imr : CONSTANT address := dua_adr + 16#17#;

dua_srb : CONSTANT address := dua_adr + 16#27#;

dua_thrb : CONSTANT address := dua_adr + 16#2F#;

TASK terminal_in IS

PRAGMA priority (1);

ENTRY char_entry;

FOR char_entry USE AT system.interrupt_vector (1);

END terminal_in;

TASK terminal_out IS

PRAGMA priority (0);

ENTRY put (item : IN character);

END terminal_out;

TASK BODY terminal_in IS

ch : character;

BEGIN

LOOP

ACCEPT char_entry DO

ch := terminal_global.the_char;

END char_entry;

text_io.put (ch);

terminal_out.put (ch);

EXIT WHEN ch = ascii.sub;

END LOOP;

assign_byte (dua_imr, 16#02#); -- disable interrupts for RxRDYB

END terminal_in;

TASK BODY terminal_out IS

ch : character;

BEGIN

LOOP

```

SELECT
  WHEN bit_value (dua_srb, 2) =>
    ACCEPT put (item : IN character) DO
      assign_byte (dua_thrb, character'pos (item));
      ch := item;
    END put;
    EXIT WHEN ch = ascii.sub;
  ELSE
    NULL;
  END SELECT;
END LOOP;
END terminal_out;

BEGIN
  define_interrupt_service_routine (isr_read'address, 5);
  assign_byte (dua_imr, 16#22#); -- enable interrupts for RxDYB
END terminal;

```

The following assembler routines also belong to the example:

```

.verstamp 2 0
.set      noreorder
.globl    ISR_READ

#--
#-----
-----
#--
#-- CONSTANT SECTION
#--
        .rdata
kseg1    =      0xA0000000
dua_adr   =      kseg1+0x1FE00000
dua_isr   =      dua_adr+0x17      #-- interrupt status register port
A
dua_rhrb  =      dua_adr+0x2F      #-- rx holding register port B
#--
        .text
#--
#-----
--
        .ent      ISR_READ
ISR_READ:
#-- Function:  Handler for RS232 Receive Interrupt from DUART (EI #5)
#--           Handles only tty1 here
#--
        sub      $sp,12

```



```

        .frame    $sp,12,$31
        .mask     -4,0x80000300
        sw        $8,0($sp)
        sw        $9,4($sp)
        sw        $31,8($sp)
#--
        lbu       $8,dua_isr
        nop
#--
        andi      $8,0x20          #-- test RxDYB bit
        beq       $8,$0,no_char    #-- nothing from tty1
        nop
#--
        lbu       $8,dua_rhrb      #-- read character into $8
        nop
#--
#-- clears interrupt pending and reset of the status bit in secport_sr
#-- is implicitly done by reading secport_rhr
#--
        sb        $8,THE_CHAR      #-- make character available to Ada
prog.
                                   #-- make interrupt entry call pending

        la        $8,_IREENTRYC
        lw        $9,0($8)
        nop
        ori       $9,1             #-- set interrupt #0 pending
        sw        $9,0($8)
no_char:
        lw        $31,8($sp)
        lw        $9,4($sp)
        lw        $8,0($sp)
        j         $31
        addu      $sp,12
        .end

```

16.6 Change of Representation

The implementation places no additional restrictions on changes of representation.

17 Appendix F: Input-Output

In this chapter we follow the section numbering of Chapter 14 of the LRM and provide notes for the use of the features described in each section.

17.1 External Files and File Objects

The implementation only supports the files `standard_input` and `standard_output` of PACKAGE `text_io`. Any attempt to create or open a file raises the exception `use_error`.

17.2 Sequential and Direct Files

Sequential and direct files are not supported.

17.3 Text Input-Output

`standard_input` and `standard_output` are associated with the RS232 serial port of the target. If the Full Target Kernel is used, then all input/output operations are done on the host using the communication line between the host and this port. Both the Debugger and the Starter are prepared to do these I/O operations.

If the Minimal Target Kernel is used, then the same serial port as in the Full Target Kernel is used, but all data of `standard_output` is directly written to this port and all data of `standard_input` is directly read from this port.

For tasking aspects of I/O operations see Chapter 14.

For further details on the I/O implementation within the Target Kernel see Chapter 19.

17.3.1 Implementation-Defined Types

The implementation-dependent types `count` and `field` defined in the package specification of `text_io` have the following upper bounds:

```
COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)
FIELD'LAST = 512
```

17.4 Exceptions in Input-Output

For each of `name_error`, `use_error`, `device_error` and `data_error` we list the conditions under which that exception can be raised. The conditions under which the other exceptions declared in the package `io_exceptions` can be raised are as described in the LRM §14.4.

NAME_ERROR
is never raised.

USE_ERROR
is raised if an attempt is made to create or open a file.

DEVICE_ERROR
is never raised.

DATA_ERROR
the conditions under which `data_error` is raised by `text_io` are laid down in LRM.

17.5 Low Level Input-Output

We give here the specification of the package `low_level_io`:

```
PACKAGE low_level_io IS
```

```
  TYPE device_type IS (null_device);
```

```
  TYPE data_type IS
    RECORD
      NULL;
    END RECORD;
```

```
  PROCEDURE send_control    (device : device_type;
```

```
        data    : IN OUT data_type);  
  
    PROCEDURE receive_control (device : device_type;  
        data    : IN OUT data_type);  
  
END low_level_io;
```

Note that the enumeration type `device_type` has only one enumeration value, `null_device`; thus the procedures `send_control` and `receive_control` can be called, but `send_control` will have no effect on any physical device and the value of the actual parameter `data` after a call of `receive_control` will have no physical significance.